EECS 351-1: Intro to Computer Graphics Project B: 3D Views & Shading

Instructor: Jack Tumblin **Winter 2019**

ADDITIONS HIGHLIGHTED IN BLUE; ~~DELETIONS IN GREY~~

Your mission in Project B is to fill the Canonical View Volume (CVV) of WebGL with the view seen by a 3D camera that users fly like an airplane, free to turn, dive, climb and go anywhere to explore a gigantic 3D 'virtual world'. Your program will automatically re-size its 3D graphics to fill the full width of your browser window and at exactly 4/5-ths (80%) of its height to show two re-sized camera views side-by-side (unlike the 2019.02.11.Cameras starter code program 7.11.JTHelloCube_Resize.html), with an orthographic view on the right, and perspective view on the left. The 3D world you explore will have patterned, grid-like 'floor' plane that stretches out to the horizon in the x,y directions (World-space +z points 'up' to the sky, ***unlike*** book starter code7.07b.JT.LookAtScene….html). Arranged on this vast floor, you will place several animated, jointed solid objects (not wireframe; not lines) that you can explore by 'flying' around, between and behind them. ~~Each vertex of each object must include its own individually-specified surface normal attributes, and these enable your shaders to compute a very basic overhead-lighting effect for diffuse/Lambertian materials.~~

You may build Project B from your Project A results, or make a new program. As with Project A, depict something *you* find interesting, meaningful and/or compelling, and use any and all inspiration sources. Perhaps some clockwork gears? An interactive NxN Rubik's cube (default is 3x3)? A steerable butterfly that flies a random path in 3D by flapping its wings, or a mechanical ornithopter? A helicopter with spinning rotors? A forest scene made from waving fractal/graftal/L-system trees and bushes (a 'tree of transformations' you can see)? Scattered wheeled vehicles, legged animals or machines, a trapeze, or perhaps a 3-wheeled car?

Requirements: **Project Demo Day (and due date): Wed Feb 20, 2019**

A)-- In-Class Demo: just like Project A, on the due date (Wed Feb 20) we all demonstrate our programs to each other in class. Two other students each evaluate your work on a 'Grading Sheet', as may Tumblin and assistants. Based on Demo Day advice, you then have ≥72 hours to revise and improve your project before submitting the final version for grading. Your grade will mix Demo Day grading sheets + your improvements.

B)-- Submit your finalized project to CMS/Canvas no more than 72 hours later (Sat Feb 23 11:59PM) to avoid late penalties. Just like Project A, submit just one compressed folder (ZIP file) that contains:

1) your written project report as a PDF file, and

2) one folder (or 'directory') that holds sub-folders with all JavaScript source code, libraries, HTML, etc. (mimic the 'starter code' ZIP-file organization) We must be able to read your report & run your program in the Chrome browser by simply uncompressing your ZIP file and double-clicking the HTML file found inside.

Please put your project report in the same directory as the HTML file.

---**IMPORTANT:** Name your ZIP file and the one folder it holds inside as: **Familyname**Personalname_ProjB
For example, my project A file would be: TumblinJack_ProjB.zip. It would contain sub-folders such as 'lib' and files such as TumblinJack_ProjB.pdf (a report), TumblinJack_ProjB.html, TumblinJack_ProjB.js ,etc.

---To submit your work, upload your ZIP file to Canvas→Assignments. DO NOT e-mailed projects (deleted!).

---BEWARE! SEVERE LATE PENALTIES! (see Canvas→Assignments, or the Syllabus/Schedule).

Project B consists of:

1)—Report: A short written, illustrated report, submitted as a printable PDF file.

Length: >1 page, and typically <5 pages, but you should decide how much is sufficient.

A complete report consists of these 3 sections:

a)--your name, netID (3 letters, 3 digits: my netID is JET861), and a descriptive title for your project
(e.g. "Project B: Flying Through a Forest of Trees", not just "Project B")

b)—a brief 'User's Guide'. Begin with a paragraph that explains your goals, then give user instructions on how to run and control the project. (*e.g.* "A, a, F, f keys rotate outer ring forwards/backwards; S, s, D, d keys rotate inner ring forwards/backwards; HUD text shows velocity in kilometers/hour.") Your classmates should be able to read ONLY this report and easily run and understand your project without your help.

c)—a brief, illustrated 'Results' section that shows **at least 4 still pictures** of your program in action (use screen captures; no need for video capture), with figure captions and text explanations. Your figure(s) also must include a sketch of your program's scene graph (the 'tree of transformations': unsure? See lecture notes 2019.01.23.VectorMathPart2_DualitySceneGraphs_VanDamm06.pdf).

2)—Your Complete WebGL program, which must include:

a)—User Instructions: When your program runs, it must explain itself to users. How? You decide! Perhaps print a brief set of user instructions below the canvas object? Or print 'press F1 for help'? Create a pop-up window? Perhaps within the 'canvas' element using the 'HUD' method in the book, or in the JavaScript 'console' window (in Google 'Chrome' browser), etc. Your program should never puzzle its users, or require your presence to explain, find, or use any of its features.

b) —'Ground Plane' Grid: Your program must clearly depict a 'ground plane' that extends to the horizon: a very large, repetitious pattern of repeated crossed lines, a 2D (or 3D) pattern of triangles, or any other shape that repeats to form a vast, flat or mostly-flat, fixed 'floor' of your 3D world. You MUST position your camera in the **x,y plane** (z=0) of your 'world-space' coordinate system, *do not* use +y' as 'up'! This grid should make any and all camera movements obvious on-screen, and form a reliable 'horizon line' when viewed with a perspective camera.

c)—Animated, adjustable, 3-Jointed, 4-segment 3D Shape: Your code must show **at least one smoothly-animated jointed 3D object** with at least four (4) sequentially-jointed parts connected by three (3, not two!!) sequential joints at different locations (one MORE joint than required for Project A). Animate those joints and enable users adjust those joint angles smoothly by interactions with the mouse and/or keyboard.

Remember, a well-designed jointed object may require you to 'push' and 'pop' matrices from your model matrix stack as you draw it. In a scene-graph, this means you have at least three sequential transform nodes; one node is a 'descendant' of another node, which in turn is a descendant of a 3rd node, and you will draw a transformed 3D part before and after you visit each of these transform nodes. A robot arm-and-hand satisfies this requirement: torso (part 1) attaches to displaced upper arm (part 2) via hinge-like shoulder (joint 1); the upper arm then attaches to displaced lower arm (part 3) through a hinge-like elbow (joint 2); lower arm then attaches to displaced hand (part 4) through a hinge-like wrist joint (joint 3). Torso movement moves all the sequentially attached parts. Conversely, you will *not* satisfy this requirement with a stick-legged starfish. If made of a pentagonal body and 5 hinged but joint-free single-segment arms, it has 6 parts and 5 joints, but no sequential joints. Adjusting one arm joint has no effect on any other part. Its scene graph holds a body transform followed by 5 children; one for each arm-angle transform, and no arm is the descendant of another arm.

d)—Four(4) or More Additional, Separate, Multi-colored Objects. 'Separate' means individually positioned, spatially separate, distinct, different-shaped objects that move differently. (For example, the top parts of a robot and the bottom parts together make up just one object, as you wouldn't move them to opposite sides of the screen). The objects don't have to move, but they do have to be distinct and fundamentally different, unrelated, spatially-separated various objects. 'Multi-color' means an object uses at least 3 different vertex colors, and WebGL must blend between these vertex colors to make smoothly varying pixel colors.

e)—Show 3D World Axes, and some 3D Model Axes: Draw one set fixed at the origin of 'world space' coordinates, and at least two others to show other coordinate systems within your jointed object. I recommend that you create a 'drawAxes()' function that draws a 3 unit-length lines: bright red for x axis, bright green for y axis, bright blue for z axis. (HINT: see quaternion starter code—it has R, G, B axis-drawing)

f)—Quaternion-based Mouse-Drag Rotation of 3D Object placed on Ground-Plane Grid. Create and draw a colorful 3D object positioned on your ground-plane grid that users can rotate intuitively and interactively by dragging the mouse on the HTML-5 canvas. Mouse-dragging should always give sensible, track-ball-like rotation results, exactly as seen in the starter code 2.09.02.06.StarterCode_Week04 →QuaternionStarter→ControlQuaternion. Unlike that starter code, your Project B mouse-drag rotations must work correctly with your movable 3D camera. Regardless of camera position, if users can see the 3D object on-screen, then they should be able to rotate it by dragging the mouse, and the rotation axis should appear to the user as perpendicular to the mouse-drag direction.

~~**f)—Demonstrate simple diffuse overhead shading** on at least one rotating/pivoting 3D shape. Each vertex of the 3D shape must contain attributes for both 'color' and 'surface-normal' vectors. Set these 'normal' vectors to unit-length, aimed outwards and perpendicular to the shape at each vertex. (HINT: surface normal attributes for a sphere are easy to find…). Then to compute the color of each vertex, your vertex shader must: 1) compute the dot-product of the normal vector and the upwards (+z) vector in the 'world' coordinate system, 2) create its non-negative 'clamped' result (if <0, set to zero; see GLSL-ES `clamp()` function) and 3) compute the vertex on-screen as: `'color'*(0.2 + 0.8*clamped_result)`. As it moves and rotates, the bottom side of your object should always look dim `(0.2*color)`, and the top should always look bright for any orientation.~~

~~**CAUTION! Don't use explicitly-defined light objects and non-diffuse materials, etc. in Project B. Why? Because we're saving them for use in Project C.**~~

g)—Two Side-by-Side Viewports in a Re-sizable Webpage: Your program must depict its 3-D scene twice, in two side-by-side viewports that together fill all the width of your browser window and 4/5ths (80%) of its height. Resizing browser window to any height or width never creates scroll-bars, empty on-screen gaps, or any distortion (stretch or squash) – by using variable, matched viewport and camera settings. The left viewport shows image from a 3D perspective camera; the right shows image from an orthographic camera.

h)—Perspective Camera AND orthographic Camera: Both camerea must use exactly the same eye point, look-at point, up vector, z-near, and z-far values. The 'perspective' camera' vertical FOV is fixed at 35° (horizontal FOV depends on browser window size), and the 'orthographic' camera's width and height must match the perspective camera's view-frustum size measured at distance z = (far-near)/3 from it's COP.

i)—View Control: smoothly & independently control 3D Camera positions and aiming direction. Both, together! Your code must enable users to explore the 3D scene via user interaction. I recommend that you use mouse dragging, arrow keys, W/A/S/D, or other widely-used key combinations to steer and move through the scene. You may design and use your own camera-movement system, but for full credit your system must allow complete 3D freedom of movement:

1. at any 3D location, your camera MUST be able to smoothly pivot its viewing direction without any change in 3D position (if you pretend that your head is the camera, you must be able to turn your head without moving your body), and:
2. your camera MUST be able to move to any 3D location from any other 3D location in one straight line, WITHOUT changing its viewing direction as it moves. You MUST NOT require users to move in only the x,y,z directions, or only in circles of varying radius! (I strongly recommend: move forward/back in viewing direction, and 'strafe' left/ right: move horizontally, perpendicular to viewing direction).

    For example: imagine a scene of 64 colorful cubes placed in a 4x4x4 grid above the 'ground plane' to form a city of floating buildings and flying cars (e.g. **http://youtu.be/IJhlD6q71YA?t=29s**). However, these streets don't follow the x,y,z directions –the 4x4x4 grid was rotated to place two opposite corners on the z axis: its streets align with vectors (1,1,1) , (-1,1,1), and (1,-1,1), and the cube-of-streets slowly tumbles; it rotates at 30 degrees/hour around an axis whose orientation also changes very slowly). Your camera must be able to'drive' down those streets easily; thread itself through all the streets and around the irregular grid of buildings, moving smoothly without any awkward zig-zagging. If your system cannot easily position the camera to 'drive around the block' in 3D, if your system rotates the camera when users move the camera, or moves the camera when users rotate the camera then it does not meet the project requirements.

BIG HINTS: If you use `LookAt()` to create your 'view' matrix, your user controls must modify BOTH the camera position (VRP or 'eye') AND the target point or 'look-at' point, and vary them independently. In class we described the 'glass-cylinder' model for camera movement that easily achieves all the Project B goals. If you do this, make global variables for eye-point, up-vector, horizontal aiming angle 'theta', and look-at point z-coordinate; compute the look-at point's x,y coordinates from eye-point and theta.

3)—Note all the opportunities for extra credit by adding more features to your project; see Grading Sheet.

Sources & Plagiarism Rules:

Simple: *never* submit the work of others as your own.
You are welcome to begin with the book's example code and the 'starter code' I supply; you can keep or modify any of it as you wish without citing its source. I strongly encourage you to always start with a basic graphics program (hence 'starter code') that already works correctly, and incrementally improve it; test, correct, and save a new version at each step.

I ***want*** you to explore -- to learn from websites, tutorials and friends anywhere (e.g. GitHub, StackOverflow, MDN, CodeAcademy, OpenGL.org, etc), and to apply what you learn in your Projects.
Please share what you find with other students, too -- list the URLs on CMS/Canvas discussion board, etc. and list in the comments the sources of ideas that helped you write your code.

BUT always, ALWAYS credit the works of others— ***no plagiarism!***

Plagiarism rules for writing essays apply equally well to writing software. You would never cut-and-paste paragraphs or whole sentences written by others and submit it as your own writing: and the same is true for whole functions, blocks and statements. ***Take their good ideas, but not their code*** add a gracious comment that recommends the inspiring source of those good ideas, and then write your own, better code in your own better style; stay

compact, yet complete, create an easy-to-read, easy-to-understand style.
Don’t waste time trying to disguise plagiarized code by rearrangement and renaming (MOSS won’t be fooled).
Instead, study good code to grasp its best ideas, learn them, and make your own version in your own style.
Take the ideas alone, not the code: make sure your comments properly name your sources.

Also, please note that I apply the ‘MOSS’ system from Stanford (https://theory.stanford.edu/~aiken/moss/) and
if I find any plagiarism evidence (sigh), the University requires me to report it to the Dean of Students for investigation.
It’s a defeat for all involved: when they find misconduct they’re very strict and very punitive.